Tera-Scale 1D FFT with Low-Communication Algorithm and Intel® Xeon Phi™ Coprocessors

Jongsoo Park[1], Ganesh Bikshandi[1], Karthikeyan Vaidyanathan[1],
Ping Tak Peter Tang[2], Pradeep Dubey[1], and Daehyun Kim[1]

[1]Parallel Computing Lab, [2]Software and Service Group, Intel Corporation

ABSTRACT

This paper demonstrates the first tera-scale performance of Intel® Xeon Phi™ coprocessors on 1D FFT computations. Applying a disciplined performance programming methodology of sound algorithm choice, valid performance model, and well-executed optimizations, we break the tera-flop mark on a mere 64 nodes of Xeon Phi and reach 6.7 TFLOPS with 512 nodes, which is 1.5× than achievable on a same number of Intel® Xeon® nodes. It is a challenge to fully utilize the compute capability presented by many-core wide-vector processors for bandwidth-bound FFT computation. We leverage a new algorithm, Segment-of-Interest FFT, with low inter-node communication cost, and aggressively optimize data movements in node-local computations, exploiting caches. Our coordination of low communication algorithm and massively parallel architecture for scalable performance is not limited to running FFT on Xeon Phi; it can serve as a reference for other bandwidth-bound computations and for emerging HPC systems that are increasingly communication limited.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming, Parallel Programming*

General Terms

Algorithms, Experimentation, Performance

Keywords

Bandwidth Optimizations, Communication-Avoiding Algorithms, FFT, Wide-Vector Many-Core Processors, Xeon Phi

1. INTRODUCTION

High-performance computing as a discipline has indeed come a long way as we celebrate the 25th Supercomputing Conference. Technology evolution often follows unpredictable paths: Power consumption and memory bandwidth have now become the leading constraints on advancing the prowess of the microprocessor, and moving data instead of computing with them dominates running time [6, 17, 27].

The foreseeable trajectories of microprocessor architectures will rely on explicit parallelism such as multi-core processors with vector instructions as well as continuing the trend of deep memory hierarchy. Explicit parallelism alleviates the need for complex architecture which consumes significant energy, and memory hierarchies help hide latency and increase bandwidth. The cost of moving data between memory hierarchies are higher, and orders of magnitude higher still between microprocessors or compute nodes connected via interconnects, even with state-of-the-art ones. It is unequivocal among researchers that interconnect speed will only deteriorate compared to compute speed moving forward.

The recurring question of whether foreseeable leading-edge microprocessor and system architectures of the time can be well utilized for applications does not lose its validity. This paper offers a case study on implementing distributed 1D FFT that affirms the effectiveness of interconnected systems of Intel Xeon Phi compute nodes. That FFT is a crucial computational method is undisputed; and unfortunately, also undisputed is the challenge in providing highly efficient implementation of it. Among FFTs, in-order 1D FFT is distinctly more challenging than the 2D or 3D cases as these usually start with each compute node possessing one or two complete dimensions of data.

We take on this 1D FFT challenge and delivered a tera-scale implementation – achieving 6.7 TFLOPS on 512 nodes of Intel Xeon Phi coprocessor. In the perspective of per-node performance, this is about fivefold better than the Fujitsu K computer [2]. This result validates the methodology we took that comprises of careful algorithm selection, a prior performance modeling, and diligent single-node architecture-aware performance optimization for key kernels. Our contributions are as follows.

(1) *We present the first multi-node 1D* FFT *implementation on coprocessors:* To the best of our knowledge, this paper is the first performance demonstration of multi-node 1D FFT on coprocessors such as Xeon Phi or GPUs [24].

(2) *We demonstrate the substantial computational advantage of Xeon Phi even for bandwidth-bound FFT:* Our implementation of low-communication FFT algorithm performs ∼2× faster when run on 512 Xeon Phi nodes than on 512 Xeon nodes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13, November 17-21 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503242

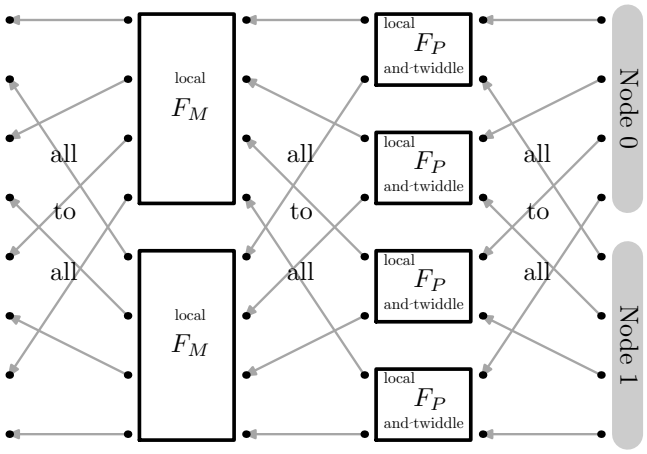


Figure 1: Cooley-Tukey Factorization [9]

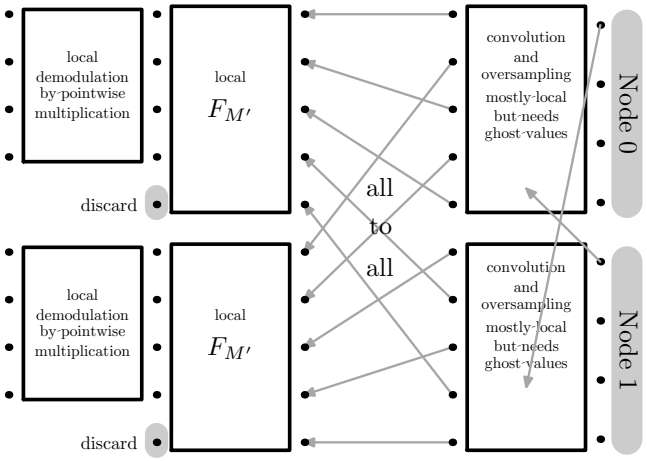


Figure 2: Segment-of-Interest Factorization [32]

(3) *We document our overall optimization methodology and specific optimization techniques:* The methodology involves performance modeling, architecture-aware optimization, and performance validation (Sections 4–6). Our FFT-motivated optimization techniques can serve as a reference for implementors of bandwidth-bound applications on Xeon Phi and other coprocessors, as our Linpack-motivated optimization techniques [15], for compute-bound programs implementors.

2. SOI FFT

This paper utilizes a relatively new FFT algorithm [32] which has a substantially lower communication cost than that of a conventional Cooley-Tukey-based algorithm. We review the key features of this low communication approach here and refer the readers to [32] for a full discussion.

A conventional algorithm [9, 18] decomposes the Discrete Fourier Transform (DFT) of $N = MP$ data points into shorter length DFTs of size M and P. Applying this decomposition recursively on these shorter length problems lead to the celebrated $O(N \log N)$ arithmetic complexity, and hence the name Fast Fourier Transform (FFT). Nevertheless, when implemented in a distributed computing environment, as depicted in Fig. 1, this method fundamentally requires three all-to-all communication steps. This all-to-all communication can account for anywhere from 50% to over 90% of the overall running time (Section 4), and was the focus of many continuing research work [5, 10, 29, 30].

The three all-to-all communication steps above can be considered as incurred by the highest level decomposition. In this context, the low communication algorithm can be understood as replacing just the highest level decomposition of a conventional FFT algorithm by an alternative, while employing standard FFT algorithms for the subsequent shorter length problems. The fundamental characteristics is that one all-to-all communication step suffices in this decomposition. In greater detail, a length $N = MP$ problem is decomposed into shorter length problems of size M' and P where M' is bigger than M by a small factor $M' = \mu M$, $\mu > 1$. The factor μ is a design choice typically chosen to be 5/4 or less (notations used in this paper is listed in Table 1). We refer to this decomposition as SOI-FFT[1]. SOI decomposition consists of three key components, as depicted in Fig. 2: (1) a convolution-and-oversampling process that involves BLAS-like computation followed by length-P FFTs, (2) one all-to-all communication step, (3) length-M' FFTs followed by demodulation with element-wise scaling.

These three steps can be expressed algebraically in terms of matrix operations involving Kronecker products. This kind of expressions has proved to be invaluable in guiding efficient implementations on modern architectures (see [11, 12, 18] for example). To compute $y = F_N\, x$, the DFT of a N-length vector x, SOI uses the formula

$$y = I_P \otimes \left(\hat{W}^{-1}\, \mathrm{P}_{\mathrm{roj}}^{M',M}\, F_{M'}\right)\, \mathrm{P}_{\mathrm{erm}}^{P,N'}\, (I_{M'} \otimes F_P)\, W\, x, \quad (1)$$

explained as follows.

(1) W is a structured sparse matrix of size $N' \times N$, $N' = \mu N$.

(2) I_m for an integer $m > 0$ stands for the identity matrix of dimension m. Given an arbitrary $J \times K$ matrix A that maps K-vectors to J-vectors: $v = Ax$, the Kronecker product $I_m \otimes A$ is a $mJ \times mK$ matrix that maps mK-vectors to mJ-vectors via

$$(I_m \otimes A) \begin{bmatrix} u^{(0)} \\ u^{(1)} \\ \vdots \\ u^{(m-1)} \end{bmatrix} = \begin{bmatrix} Au^{(0)} \\ Au^{(1)} \\ \vdots \\ Au^{(m-1)} \end{bmatrix}$$

Expressions of the form $I_m \otimes A$ are naturally parallel.

[1] See [32] for the rationale of this name.

Table 1: Notations used in this paper

N	The number of input elements
P	The number of compute nodes
$M = \frac{N}{P}$	The number of input elements per node
$\mu = \frac{n_\mu}{d_\mu}$	The over sampling factor (typically $\leq 5/4$)
$N' = \mu N, M' = \mu M$	
W	matrix used in convolution-and-oversampling
B	the convolution width with typical value 72

(3) $\mathrm{P}_{\mathrm{erm}}^{\ell,n}$ where ℓ divides n denotes the stride-ℓ permutation of an n-vector: $w = \mathrm{P}_{\mathrm{erm}}^{\ell,n} v \Leftrightarrow v_{j+k\ell} = w_{k+j(n/\ell)}$, for all $0 \le j < \ell$ and $0 \le k < n/\ell$. The term $\mathrm{P}_{\mathrm{erm}}^{P,N'}$ in Equation 1 is the one all-to-all communication step required by SOI.

(4) $\mathrm{P}_{\mathrm{roj}}^{m',m}$ for $m' \ge m$ is the $m \times m'$ matrix that takes a m'-vector and returns the top m elements.

(5) $\hat{W}$ and $\hat{W}^{-1}$ are $M \times M$ diagonal and invertible matrices. Action of $\hat{W}^{-1}$ in Equation 1 corresponds to demodulation with element-wise scaling.

Equation 1 encapsulates all the important features involved in an implementation. This equation is the starting point of performance modeling (Section 4). The Kronecker product notation conveys parallelism explicitly. For example, $I_p \otimes A$ expresses that p instances of the operator A are executed in parallel, each on a node. Moreover, because $I_{\ell \times m} \otimes A = I_\ell \otimes (I_m \otimes A)$, the operation such as $I_{M'} \otimes F_P$ can be realized as $I_P \otimes (I_{M'/P} \otimes F_P)$, suggesting that M'/P instances of F_P are executed on one node, offering node level parallelism. We use a hybrid parallelization scheme, where MPI is used for inter-node parallelization and OpenMP is used for intra-node parallelization. Equation 1 reveals the key steps that are targets for optimization: the all-to-all step $\mathrm{P}_{\mathrm{erm}}^{P,N'}$ (Section 5.1), the large local FFT $F_{M'}$ (Section 5.2), and the convolution step, which is applying W to the input data (Section 5.3).

3. INTEL XEON PHI COPROCESSOR

A wave of new HPC systems has been emerging that take advantage of massively parallel computing hardwares such as Xeon Phi coprocessors and GPUs [24]. When their abundant parallelism is well utilized without their memory bandwidth capacity exceeded, these many-core processors with wide-vector instructions can provide an order-of-magnitude higher compute power than traditional processors. This section describes the distinctive architectural performance features of the Xeon Phi coprocessor and highlights the key considerations we take to deliver our tera-scale FFT implementation.

Intel Xeon Phi coprocessors are the first commercial product of the Intel® MIC architecture family, whose specification is compared with a dual-socket Xeon E5-2680 in Table 2. It is equipped with many cores, each with wide-vector units (512-bit SIMD), backed by large caches and high memory bandwidth. In order to maximize efficiency in power as well as area, these cores are less aggressive: they execute instructions in-order and run at a lower frequency. Each Xeon Phi chip can deliver a peak 1 TFLOPS double-precision performance, approximately 6× than a single Xeon E5 processor. Unlike GPUs, Xeon Phi executes the x86 ISA, allowing the same programming model as conventional x86 processors. The same software tools such as compilers and libraries are available for both host Xeon processors and Xeon Phi coprocessors.

Table 2: Comparison of Xeon and Xeon Phi

	Xeon E5-2680	Xeon Phi SE10
Socket×core×SMT×SIMD	2 × 8 × 2 × 4	1 × 61 × 4 × 8
Clock (GHz)	2.7	1.1
L1/L2/L3 Cache (KB)*	32/256/20,480	32/512/-
Double-precision GFLOP/s	346	1,074
STREAM bandwidth [15, 19]	79 GB/s	150 GB/s
Bytes per Ops	0.23	0.14

*Private L1/L2, shared L3

For Xeon Phi coprocessors, optimizing data movement is particularly important for running bandwidth-bound local FFTs (denoted as $F_{M'}$ in Fig. 2 and Equation 1). Even though Xeon Phi provides memory bandwidth higher than traditional processors, its compute capability is even higher: i.e., its bytes per ops ratio (*bops*) is lower as shown in Table 2. We take advantage of Xeon Phi's large caches to reduce the main memory accesses, applying various locality optimizations (Sections 5.2 and 5.3).

We pay close attention to the PCIe bandwidth in order to realize scalable performance with multiple coprocessors. Each compute node is composed of a small number of host Xeon processors and Xeon Phi coprocessors connected by PCIe interface, which typically sustains up to 6 GB/s bandwidth. We can use Xeon Phi by offloading compute intensive kernels from the host (*offload mode*) or by running independent MPI processes (*symmetric mode*). FFT can be called from applications written in either mode. Although this paper focuses on symmetric mode, most of the optimizations presented are applicable to both modes. An exception is optimizations of direct MPI communication between Xeon Phis for effectively overlapping data transfers over PCIe with transfers over InfiniBand (Section 5.1). These PCIe related optimizations in symmetric mode have not been discussed as often as those in offload mode, which can be found, for example in [15, 25]. Section 7 will further compare symmetric and offload modes.

4. PERFORMANCE MODELING

This section develops a model that projects the performance improvement of SOI FFT from using coprocessors. It shows, among other things, that SOI FFT can run about 70% faster on Xeon Phi coprocessors as it can on the same number of dual-socket Xeon E5-2680 nodes.

As a function of the input size, N, let $T_{\mathrm{fft}}(N)$ and $T_{\mathrm{conv}}(N)$ be the execution time of node-local FFT and convolution computations, and $T_{\mathrm{mpi}}(N)$ be the latency of one all-to-all exchange of N data points.

The execution time of SOI without using coprocessors can be modeled as

$$T_{\mathrm{soi}}(N) \sim T_{\mathrm{fft}}(\mu N) + T_{\mathrm{conv}}(N) + \mu T_{\mathrm{mpi}}(N).$$

Compare this with the execution time of a conventional FFT algorithm that uses Cooley-Tukey factorization (a representative is MKL FFT):

$$T_{ct}(N) \sim T_{\mathrm{fft}}(N) + 3T_{\mathrm{mpi}}(N).$$

In the symmetric mode, the execution time of SOI can be modeled as

$$T_{\mathrm{soi}}^{\phi_{\mathrm{sym}}}(N) \sim T_{\mathrm{fft}}^{\phi}(\mu N) + T_{\mathrm{conv}}^{\phi}(N) + \mu T_{\mathrm{mpi}}(N).$$

T_{fft}^{ϕ} and T_{conv}^{ϕ} are the execution times of node-local FFT and convolution computations on Xeon Phi coprocessor, respectively (ϕ refers to Xeon Phi).

Each component of execution time can be computed as follows once parameters such as bandwidth and compute

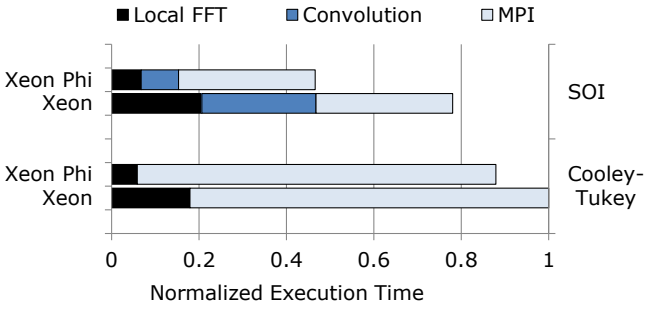


Figure 3: Estimated performance improvements from our performance model. The execution time is normalized to that of running an FFT algorithm with Cooley-Tukey factorization in 32 nodes of dual-socket Xeon E5-2680 processor.

efficiency are available:

$$T_{\text{fft}}(N) = \frac{5Nlog_2N}{\text{Efficiency}_{\text{fft}} \cdot \text{FLOPS}_{peak}},$$
$$T_{\text{conv}}(N) = \frac{8B\mu N}{\text{Efficiency}_{\text{conv}} \cdot \text{FLOPS}_{peak}},$$
$$T_{\text{mpi}}(N) = \frac{16N}{\text{BW}_{\text{mpi}}}.$$

Sections 5.2 and 5.3 explain the number of floating point operations in the node-local FFT and convolution, $5Nlog_2N$ and $8B\mu N$ (B denotes the convolution width with typical value 72). We assume inputs are double-precision complex numbers (i.e., 16 bytes per element).

Let us instantiate our performance model with realistic parameters to assess potential performance gain from using coprocessors. We assume compute efficiency in local FFT and convolution ($\text{Efficiency}_{\text{fft}}$ and $\text{Efficiency}_{\text{conv}}$) to be 12% and 40%, both in Xeon and Xeon Phi. Section 6 shows that these are the actual efficiencies that we achieve on Xeons, and Sections 5.2 and 5.3 present optimizations that allow us to achieve comparable efficiencies on Xeon Phi. Since a single Xeon Phi coprocessor has ∼3× peak FLOPS than a dual socket Xeon, T^{ϕ}_{fft} and T^{ϕ}_{conv} are $\sim\frac{1}{3}$ of T_{fft} and T_{conv}, respectively. Note that our performance model assumes that MPI bandwidth between Xeon Phis is the same as that between Xeons, and this is achieved by optimizations described in Section 5.1. The oversampling factor μ, which is relevant when comparing SOI to Cooley-Tukey, is set at 5/4.

Consider 32 compute nodes and $N = 2^{27}\cdot 32$ (similar to the input size used in our evaluation presented in Section 6). We assume 3 GB/s per-node MPI bandwidth: i.e., the aggregated bandwidth, BW_{mpi} is 32×3. Then, T_{fft}=0.50 sec., T^{ϕ}_{fft}=0.16, T_{conv}=0.64, T^{ϕ}_{conv}=0.21, and T_{mpi}=0.67.

Fig. 3 shows the estimated running time of both Cooley-Tukey and SOI FFT in Xeon and Xeon Phi. With SOI algorithm, Xeon Phi achieves nearly 70% speedup over Xeon. The additional computation introduced by SOI FFT is offset by high compute capability of Xeon Phi. On the other hand, with the standard Cooley-Tukey algorithm, Xeon Phi yields only 14% speedup. This is because the large communication time in Cooley-Tukey algorithm is the limiting factor in speeding up FFT with coprocessors.

5. PERFORMANCE OPTIMIZATIONS

As a fundamental mathematical function, FFT has been optimized, deservedly so, on many specific processor architectures for a long time (see for example [11, 13, 28] and the many references thereof). It is high time we applied rigorous and architecture-aware optimizations on the relatively new Xeon Phis coprocessors. We discuss three components of our optimizations—direct MPI communication between Xeon Phis coprocessors, large node-local FFTs, and convolution-and-oversampling. For the last two, we focus on bandwidth related optimizations although thread-level parallelization and vectorization are also critical and non-trivial. This is because what clearly distinguishes FFT from other fundamental kernels such as Linpack is its bandwidth bounded nature.

5.1 MPI All-to-All: $\text{P}^{P,N'}_{\text{erm}}$

A novel feature of Xeon Phi architecture and software ecosystem is that, in the symmetric mode, threads can make direct MPI calls, freeing up the user from orchestrating data transfers between the host and the coprocessor.

The SOI FFT algorithm requires two communication steps—nearest neighbor communication before convolution (depicted as two right-most arrows in Figure 2) and all-to-all communication. The nearest neighbor communication transfers short messages (tens of KBs) and is bound by latency. Xeon Phi's native MPI is optimized well for such latency bound short messages. The all-to-all step transfers long messages (several MBs), which is bottlenecked by the available interconnect bandwidth. The native MPI was found to be inefficient in handling long message transfers. To overcome this, we use a reverse communication MPI proxy layer described in [16] for the all-to-all.

The proxy layer dedicates one core (from the host) to process requests from local Xeon Phi coprocessors. The host core extracts the data from the Xeon Phi memory to host memory via direct memory access (DMA) and sends the data to the destination node using remote-DMA over InfiniBand. Similarly, at the destination, the host core receives the data and copies it directly to the Xeon Phi memory. Synchronization between the host and Xeon Phi co-processors are performed by handshakes using control messages. The control messages are stored in a memory mapped queue shared by both host and the co-processor [16].

PCIe transfer times from Xeon Phis to the host is hidden by pipelining with InfiniBand transfers. The application data are split into several chunks to be pipelined, and the chunk size is appropriately chosen to balance the latency and throughput: e.g., smaller chunk sizes keeps latency low but at the cost of lower throughput.

5.2 Large Local 1D FFT: $F_{M'}$

5.2.1 Memory Bandwidth Constraints

Recall that SOI FFT is really a different factorization (Equation 1) applied at the highest level. The subsequent local DFTs can be handled by the traditional Cooley-Tukey approach which recursively decomposes a large local 1D DFT into smaller ones. Fig. 1 depicts Cooley-Tukey factorization applied to multi-node settings, but its application to local 1D FFTs is conceptually the same[2]. Not only does Cooley-Tukey formulation reduce the number of operations from

[2]Our SOI factorization can also be recursively applied to local FFTs,

```
1 transpose P×M to M×P    // 1 load, 1 store
2 M P-point FFTs          // 1 load, 1 store
3 twiddle multiplication  // 2 load, 1 store
4 transpose M×P to P×M    // 1 load, 1 store
5 P M-point FFTs          // 1 load, 1 store
6 transpose P×M to M×P    // 1 load, 1 store
```

(a) A Naïve 2D 6-step implementation with 13 memory sweeps. The input arranged in a 2D matrix is explicitly transposed so that P-point and M-point FFTs can be operated on contiguous memory regions. Row-major order as in C is assumed.

```
  // steps 1-4:  1 load, 1 store
  loop_a over M columns, 8 columns at a time
1   copy P × 8 to a contiguous buffer
2   8 P-point FFTs together in SIMD
3   twiddle multiplication with smaller tables
4   permute and write back
  // steps 5-6:  1 load, 1 store
  loop_b over P rows, 8 rows at a time
5   8 M-point FFTs one-by-one
6   permute and write back
```

(b) An optimized 6-step implementation with 4 memory sweeps, where loops are fused and smaller twiddle coefficient tables are used.

Figure 4: Local FFT pseudo code

$O(N^2)$ to $O(NlogN)$, but recursive invocation of the algorithm for smaller FFTs also reduces the number of memory sweeps[3] from $O(N)$ to a small constant in cache-based architectures. Nevertheless, 1D FFTs are still typically memory bandwidth bound in contrast to other computations with higher arithmetic intensity (e.g., matrix multiplication with compute complexity $O(N^3)$).

FFT for N complex numbers has $\sim 5Nlog_2N$ floating-point operations, assuming a radix-2 FFT implementation. Let us first consider a type of 1D FFT that is least memory bound, where the input, output, and scratch data all together fit in on-chip caches. Thus, there is one memory read and one memory write of the entire data. For example, a 512-point double-precision complex FFT has $5 \cdot 512 \cdot log_2 512$ floating point operations, and $2 \cdot 512 \cdot 16$ bytes are transferred from/to memory. The communication-to-computation ratio measured in bytes per ops (*bops*) is about 0.7. The machine bops of a dual-socket Xeon E5-2680 running at 2.7 GHz is 0.23 as shown in Table 2, which is considerably smaller than the algorithmic bops, thus making the performance of FFT bound by memory bandwidth. The gap between the machine and algorithmic bops is even wider in Xeon Phi: the machine bops of Xeon Phi is 0.14 as shown in Table 2. Assuming that compute is completely overlapped with memory transfers, the maximum achievable compute efficiency is only $\frac{0.14}{0.7} = 20\%$. Our highly tuned small sized FFT implementations achieve close to 20% efficiency, confirming this theoretical projection. For larger 1D FFTs, even achieving this 20% bound is a challenge because additional memory sweeps are required, and the memory accesses will be in larger strides, sometimes greater than a page size.

but communication-to-computation ratio is much higher within a compute node, where additional computation of SOI is harder to be compensated.

[3] One memory sweep refers to loading or storing the entire N data points.

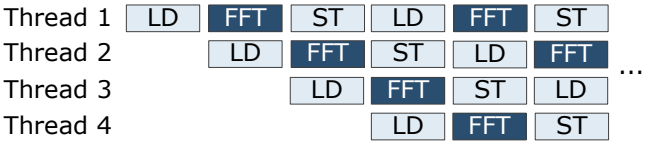


Figure 5: Overlapping compute with memory transfers on co-processors within a node

5.2.2 *Bandwidth-Efficient 6-step Algorithm*

Several bandwidth-efficient implementations of Cooley-Tukey factorization have been proposed for large 1D FFTs. One of them is the 6-step algorithm with 2D decomposition by Bailey [5] whose naïve implementation is shown in Fig. 4(a). Here, the 1D input vector is organized into a 2D matrix of size $P \times M$. Steps 1, 4, and 6 correspond to all-to-all communications in Fig. 1, steps 2 and 3 correspond to boxes on the right side with label F_P, and step 5 corresponds to boxes on the left side with label F_M. Although the all-to-all communications can be implicitly performed by strided memory access in shared memory environment within a compute node, the 6-step algorithm explicitly transposes the data so that P-point and M-point FFTs can be performed on contiguous memory regions. This greatly reduces TLB and cache conflict misses.

Bailey also presents a variation of his algorithm shown in Figure 4(b), where loops are fused so that memory access is reduced significantly [5]. For example, instead of performing P-point FFTs for the entire data writing outputs to the memory, step 2 can be stopped after a small number of columns and, then, step 3, twiddle multiplication, can be started reading the FFT results from on-chip caches. We can also reduce the size of twiddle coefficient tables at the expense of slightly more computation by exploiting that a twiddle factor $\exp\left(\frac{\iota 2\pi(k_1+k_2)}{N}\right)$ equals $\exp\left(\frac{\iota 2\pi k_1}{N}\right) \cdot \exp\left(\frac{\iota 2\pi k_2}{N}\right)$. This optimization is called dynamic block scheme [5].

5.2.3 *Architecture-Aware Bandwidth Optimizations*

Even though the optimized 6-step algorithm significantly reduces the memory bandwidth requirement of large 1-D FFTs, numerous architecture-aware optimizations are desired for high-performance FFT implementations, especially for architectures with lower bandwidth-to-computation ratio such as Xeon Phi.

Hiding Memory Latency.

For each P-point or M-point FFTs, we copy inputs to a contiguous buffer, compute the FFTs, and copy the buffer back to memory. These three stages are executed in a pipelined manner with 4 simultaneous multiple threads (SMTs) per core as shown in Fig. 5.

The memory latencies during the copy is hidden by software prefetch instructions. The contiguous buffers are sized to fit L2 so that the FFT stage requires L1 prefetch instructions only.

The contiguous buffer is padded to avoid cache conflict misses. When the buffer is copied back, non-temporal stores are used to save bandwidth. A normal store loads a cache

line from the memory, modifies the line, and writes back the line, generating two transfers. A non-temporal store only writes a cache line to main memory without allocating a line, reducing the number of transfers to one.

Reducing Working Set by Fine-grain Parallelization.

In a simple parallelization scheme, each thread would independently work on individual P-point or M-point FFTs. However, the FFT data sometimes cannot fit the LLC; e.g., suppose M=32K, then 32K double precision complex numbers already occupy the entire capacity of an L2 cache in Xeon Phi (512 KB).

To avoid memory accesses from overflowing the LLC, we use a finer grain parallelization scheme, where multiple cores collaboratively work on a single FFT. As a trade-off, this leads to core-to-core communication and more synchronization. We carefully design so that only one core-to-core global "read" is required per FFT (no "writes"). We measure the overhead of the inter-core read to be smaller than the overhead we would pay when the data involved in FFT overflows the LLC.

An alternative is 3D decomposition that reduces the size of individual FFTs need to be performed. For example, instead of decomposing an 1G-point FFT into two groups of 32K 32K-point FFTs, we can have a 3D decomposition with three groups of 1M 1K-point FFTs. However, this 3D decomposition requires 2 extra memory sweeps, which are measured to have more overhead than one core-to-core read required in our fine-grain parallelization scheme.

5.2.4 Other Optimizations

Vectorization.

Our implementation internally uses "Struct of Arrays" (SoA) layout for arrays with complex numbers that avoids gather and scatter or cross-lane operations. The interface also supports "Array of Structs" (AoS) to increase MPI packet lengths by sending reals and imaginaries together. The longer packet length is advantages in sustaining the MPI bandwidth with many nodes.

Step 2 performs FFTs in strides of P. We vectorize this step by performing vector-width (i.e., 8) independent FFTs as shown in Fig. 4(b) (*outer-loop vectorization*). Step 5 performs unit-stride FFTs on a row, and we use *inner-loop vectorization* for this.

Step 6 performs global permutation of the input, which involves transpositions of 8×8 arrays with double-precision numbers. Each transposition requires 8 vector loads and 64 stores or 64 loads and 8 vector stores, total 72 memory instructions. We reduce the number of memory instructions required to 48 (32 loads and 16 stores) using cross-lane load/store instructions provided in Xeon Phi. The cross-lane instructions load/store contiguous values in a cache line to/from discontinuous lanes in vector registers [1]. Xeon Phi also provides gather/scatter instructions that can be used for transposition, but `load_unpack` and `store_pack` deliver better performance. This transposition can be used in many cases, including local permutations before global all-to-alls in SOI algorithm ($\mathrm{P}_{\mathrm{erm}}^{P,N'}$ in Equation 1 involves local permutation followed by an all-to-all communication).

Register usage and ILP Optimizations.

Xeon Phi has 32 general purpose 512 bit registers. To ensure optimal register utilization, we use radix 8 and 16, case by case. We unroll the leaf of the FFT recursion to exploit the instruction-level parallelism. About 12% of the operations are replaced with fused-multiply-add instruction supported on Xeon Phi.

Saving Bandwidth by Fusing Demodulation and FFT.

This optimizations is specific to the local FFT used in SOI algorithm. The demodulation step in SOI FFT requires multiplying the output of local FFT with a window function, $\hat{W}^{-1}$ described in Section 2. As a separate stage, this requires 3 memory sweeps—1 read of array, 1 read of the window function constants and 1 write to array. We save two of the sweeps by fusing this computation with the step 5 of local FFT shown in Fig. 4(b).

5.3 Convolution-and-Oversampling: Wx

The arithmetic operations of local FFTs in our low communication algorithm is comparable to that of standard FFT implementations such as the one in MKL. One can view that the arithmetic operations incurred in convolution-and-oversampling (or convolution in short) is the extra arithmetic cost that SOI algorithm incurs in order to reduce communication cost. Optimizing convolution is important.

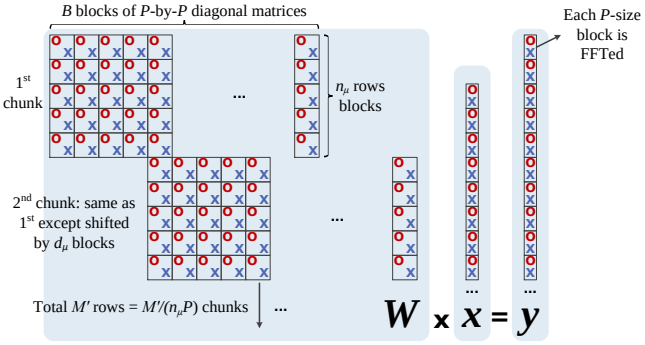
The convolution step multiplies the input with matrix W described in Section 2. This multiplication is carried out by each node in parallel, performing a local matrix-vector product. The local matrix is highly structured, depicted in Fig. 6(a). The computation loop on a node consists of $M'/(n_\mu P)$ chunks of operations, each of which is $n_\mu P$ length-B inner products. After each chunk of operation, the input is shifted by $d_\mu P$.

Each length-B inner-product of complex numbers involves B complex multiplications and $B-1$ complex additions, which amounts to $6B+2(B-1) \sim 8B$. Therefore, the number of floating point operations in the convolution step is $8B\mu N$. With $N = 2^{27} \times 32$, $B = 72$, and $\mu = \frac{8}{7}$, the convolution step has about 5× floating point operations compared to the local FFT. Fortunately, in contrast to 1D local FFT, the convolution step has a lower communication-to-compute ratio (i.e., lower bops). Therefore, the convolution step can achieve ∼4× of compute efficiency than the local FFT (40% vs. 12%) both in Xeon and Xeon Phi, leading to similar execution times.

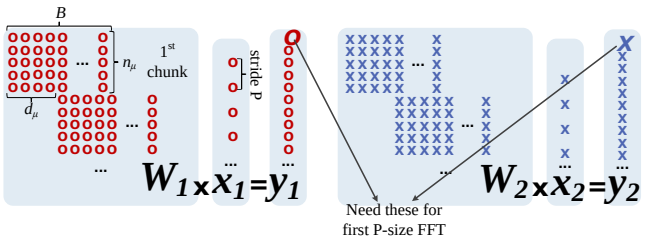
We list standard optimizations applied [33]. We could use a BLAS library to implement the matrix-vector multiplication, Wx, but our optimizations exploit the structure of W, achieving a higher efficiency. The same set of optimizations are applied to and are beneficial in both Xeon and Xeon Phi.

Reducing Working Set by Loop Interchange.

A straightforward implementation of the matrix vector multiplication shown in Fig. 6(a) simply performs inner-product per row, while skipping empty elements in the matrix. Parallelization is applied by distributing chunks of rows to threads. A key bottleneck in this implementation is that its performance degrades with more nodes. The number of distinct elements in local matrix W, $n_\mu PB$, is proportional to P, the number of nodes. All of these $n_\mu PB$ elements are accessed during a chunk of operations for $n_\mu P$ rows, overflowing the LLCs and incurring a large number of cache


B blocks of P-by-P diagonal matrices
Each P-size block is FFTed
1st chunk
n_μ rows blocks
2nd chunk: same as 1st except shifted by d_μ blocks
Total M′ rows = M′/(n_μP) chunks
W x x = y

(a) The matrix is structured as shown, where each P-by-P block is a diagonal matrix. Here, P=2, n_μ=5, d_μ=4, and the oversampling factor μ=n_μ/d_μ=5/4. The same chunk repeats while shifting, and we compactly store only the distinct $n_\mu PB$ elements. Still, the size grows as we add nodes, incurring more cache misses.



(b) The matrix multiplication Wx can be decomposed into P independent smaller multiplications since each P-by-P block is diagonal. The corresponding pseudo code is shown in Fig. 7.

Figure 6: Convolution-and-oversampling Wx on a node (a) in its original form and (b) in its decomposed form.

misses. This is particularly problematic in Xeon Phi with private LLCs[4].

To overcome this, we organize the matrix vector multiplication differently as shown in Fig. 6(b). We decompose $W \times x$ into P multiplications of sub-matrices and sub-vectors—the second sub-matrix W_2 has $n_\mu B$ distinct elements from $(2, 2)^{th}$ elements of each P-by-P blocks, depicted as "X"s in Fig. 6(b). By operating on these smaller independent matrix-vector multiplications one by one, we can keep the working set size constant regardless of the number of compute nodes.

This can be viewed as an application of loop interchange optimization [33]. Fig. 7 shows pseudo code of the optimized convolution implementation. In the straightforward implementation, `loop_a-c` used to be a single loop that iterates over rows. We tile the loop into three, where `loop_b` is the outer-most and `loop_a` is the inner-most. Iterations of `loop_a` are independent because P-by-P blocks are diagonal matrices. Therefore, we can *interchange* the loop order so that `loop_a` becomes the outer-most as shown in Fig. 7. We also apply the thread-level parallelization to `loop_a` since there is no data sharing between its iterations, which matches well with Xeon Phi's private LLCs.

[4]The matrix elements are duplicated in each private LLC whose capacity is 512 KB. On the other hand, in the Xeon processor, the elements are stored in a large shared LLC (20 MB in Xeon E5-2680).

```
loop_a over P sub-matrices/vectors
        // thread-level parallelization
  loop_b over M′/(n_μ P) chunks
    loop_c over n_μ rows
      loop_d over B // inner product
```

Figure 7: Pseudo code of optimized convolution implementation

In the straightforward implementation, as indicated by Fig. 6(a), once P rows are available, we can immediately start a P-point FFT (denoted as F_P in Equation 1), saving one memory sweep. This can be viewed as a loop fusion optimization similar to the one used in the optimized 6-step local FFT (Section 5.2). However, this optimization cannot be applied to the decomposed form shown in Fig. 6(b) since even the first P outputs are available only after all P multiplications of sub-matrices and sub-vectors are finished. Nevertheless, the decomposed form leads to an implementation scalable for large scale clusters. The overhead of the extra main memory sweep can be mitigated by using non-temporal store instructions, similarly to using non-temporal stores when writing buffers back to memory in local FFTs (Section 5.2). Section 6 shows that, when non-temporal stores are used, the decomposed form yields a competitive performance to that of the straightforward implementation even with a small number of nodes.

Avoiding Cache Conflict Misses by Buffering.

A consequence of the loop interchange described above is conflict misses from long-stride access to input. In the decomposed form shown in Fig. 6(b), inputs are accessed in stride of P. When P is a large power of two, only a few cache sets are utilized.

To address this, we stage the input through contiguous locations. As can be seen from Fig. 6(b), B input elements are accessed n_μ times while one chunk of the submatrix is processed. After finishing one chunk, the input is shifted by d_μ. We maintain a circular buffer that holds B elements and copy d_μ elements from the input to the buffer for each iteration of `loop_b` that processes one chunk. Within the iteration of `loop_b`, inputs are accessed through this buffer, translating long-stride accesses to contiguous ones. More precisely, we translate B non-contiguous loads to B contiguous loads, d_μ non-contiguous loads, and d_μ contiguous stores. Since B is sufficiently larger than d_μ (typical values are 72 and 4), a large fraction of non-contiguous accesses can be eliminated and the overhead associated with extra loads and stores are minimal.

This optimization resembles the memory latency hiding optimization presented in Section 5.2 and also the buffering optimization in architecture-friendly radix sort implementation presented in [26]. In both cases, contiguous buffers that fit in small caches are maintained, and copies to the buffer is optimized with software prefetch instructions.

Vectorization.

For efficient vectorization and a better spatial locality, inputs in a cache line need to be accessed together. Therefore, we apply loop tiling optimization [33] to `loop_a`, creating `loop_a1` and `loop_a2`. The inner loop, `loop_a2`, iterates

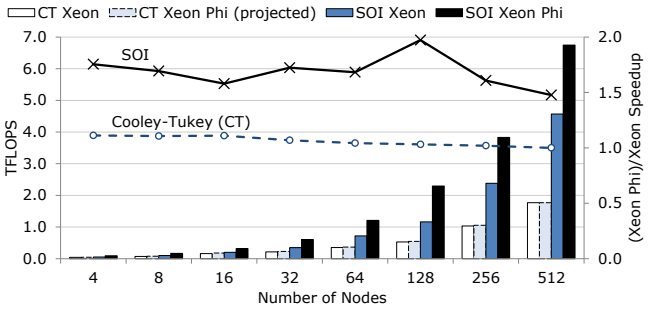


Figure 8: Weak scaling FFT performance ($\sim 2^{27}$ double precision complex numbers per node). The bar graph shows the best performance among multiple runs in TFLOPS calibrated by the left y-axis. The line graphs, calibrated by the right y-axis, show speed-ups of Xeon Phi over Xeon, when Cooley-Tukey factorization is used and when SOI is used.

over a single cache line with 8 double-precision numbers. We make `loop_a2` the inner-most one, and apply vectorization to it.

Exploit Temporal Locality by Loop Tiling and Unroll-and-Jam.

Iterations of `loop_b` reuse the same matrix elements; `loop_b` iterates over matrix chunks and the chunks actually refer to the same elements in the compact representation of matrix W. As shown in Fig. 6(b), n_μ iterations of `loop_c` reuse the same input elements. To exploit the temporal locality in caches, we tile the loops, creating `loop_b1`, `loop_b2`, `loop_c1`, and `loop_c2`. Then, we interchange loops so that `loop_b2` and `loop_c2` can be inside `loop_d`. To exploit the temporal locality in the register level, we also unroll `loop_b2` and `loop_c2` a few times and jam them into the inner-most loop. With this unroll-and-jam optimization, a matrix/input value loaded to a register is reused for multiple inner-products.

6. EVALUATION

Experiments are run on a cluster named Stampede which is part of the computing infrastructure within Texas Advanced Computing Center, whose system configuration is listed in Table 3. Section 6.1 presents the overall performance and scalability, and Sections 6.2 and 6.3 present detailed performance analyses on local FFT and convolution steps, respectively.

6.1 Overall Performance and Scalability

We achieve 6.7 TFLOPS with 512 nodes, each of which runs one Xeon Phi card, as shown in Fig. 8. Let us put this performance in the context of HPC challenge benchmark (HPCC) results as of April 2013 [2]. The highest global FFT performance (G-FFT) is 206 TFLOPS in Fujitsu K computer [3] with 81K compute nodes. We achieve $\sim 5\times$ per-node performance. This is a significant result considering that K computer uses a custom Tofu interconnect with 6D torus topology, while Stampede uses relatively common FDR InfiniBand with a fat-tree topology. The per-node performance is an important metric for communication bound 1D-FFT because a higher per-node performance leads to the same performance

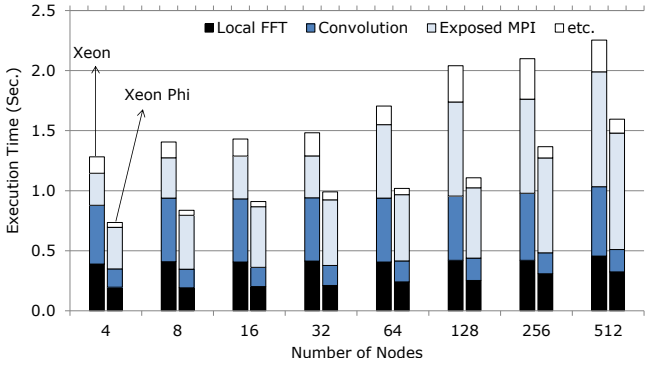


Figure 9: Execution time breakdowns of SOI algorithm.

with fewer nodes, hence less dependent on non-linear scaling of aggregated interconnect bandwidth. Nevertheless, the K computer result is with a considerably larger number of nodes, and it remains as future work to show scalability of our implementation to a similar level.

Fig. 8 plots the weak-scaling performance of MKL FFT, a repsentative of optimized Cooley-Tukey implementations, running on Xeon (`CT Xeon`), SOI FFT running on Xeon (`SOI Xeon`), and SOI FFT running on Xeon Phi (`SOI Xeon Phi`). We also project the performance of an optimized implementation of Cooley-Tukey factorization running on Xeon Phi (`CT Xeon Phi`)[5]. Since all-to-all MPI communication accounts for a major fraction of execution time in Cooley-Tukey factorization implementations, using Xeon Phi provides marginal performance improvements, $\sim 1.1\times$. This matches with the estimated performance improvements derived in Section 4. The communication requirement is reduced by SOI algorithm, thereby having a considerably higher

[5]The execution time is estimated by, as in Section 4, $T^{\phi}_{ct} = T^{\phi}_{\text{fft}} + 3T_{\text{mpi}}$, where T^{ϕ}_{fft} is from our node local FFT performance measurements and T_{mpi} is from an all-to-all MPI bandwidth benchmark.

Table 3: Experiment Setup on Stampede Cluster*

Processor	See Table 2 for Xeon and Xeon Phi specs.
PCIe BW	6 GB/s
Interconnect	FDR InfiniBand, A two-level fat tree
Compiler	Intel® Compiler v13.1.0.146
MPI	Intel® MPI v4.1.0.030 2 processes per node (Xeon), 1 per node (Xeon Phi)
SOI	8 or 2 segments/process, μ=8/7
MKL	v11.0.2.146

*Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.
Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

speed-up by using Xeon Phi (1.5–2.0×). This again is similar to the 1.7× speed-up estimated in Section 4.

Fig. 9 provides an in-depth view on time spent by each component. Xeon and Xeon Phi achieve a similar compute efficiency, 12%, in large local M'-point FFTs with 512 nodes. More detailed analysis on local FFT performance in Xeon Phi is presented in Section 6.2. The compute efficiency of the convolution step in Xeon and Xeon Phi are 42% and 38%, respectively, in 512 nodes. The convolution time does not increase with more nodes due to the loop interchange optimization that keeps the working set size constant (Section 5.3). A large fraction of `etc.` in Xeon is from demodulation step. Since we use out-of-the-box MKL library on Xeon, the demodulation step is not merged with local FFT on Xeon.

The time spent on MPI communication slowly increases with more nodes, which indicates that the interconnect is not perfectly scalable. Even though the same interconnect is used on Xeon, the exposed MPI communication time is larger in Xeon Phi because less communication can be overlapped due to faster computation.

Although one segment per MPI process is assumed so far for brevity, multiple segments can be used per process. Using multiple segments allows all-to-all communications to be overlapped with M'-point FFTs and demodulation. Suppose that we have 4 segments in Fig. 2, 2 per process. After all-to-all for the first segment in each process, we can overlap the second all-to-all with M'-point FFTs and demodulation step of the first segment.

Our evaluation uses 8 segments per MPI process for $\leq$128 nodes and 2 segments per MPI process for $\geq$ 512 nodes. In the all-to-all communication of cluster-scale 1D-FFTs, the amount of data transferred between a pair of node decreases in proportion to the number of nodes in weak scaling scenarios. This results in shorter packets in large clusters, which is a challenge for sustaining a high MPI bandwidth. Using fewer segments per node can mitigate by increasing the packet length. Although this reduces the opportunity of overlapping the communication with computation, this is a trade-off worth making in already communication dominated settings with many nodes.

Several FFT implementations using Cooley-Tukey factorization have used similar overlapping [10], but the impact of the overlapping is bigger in SOI because communication and computation times are more balanced. In addition, multiple segments will be useful for load balancing heterogeneous processes. For example, we can assign 1 segment per a socket of Xeon E5-2680 and 6 segments per Xeon Phi (recall that a Xeon Phi has ∼6× compute capability). As can be seen here, the number of segments per process is an useful parameter, which makes the improvements in convolution scalability with respect to the number of segments more important (Section 6.3)

6.2 Large Local FFT

Fig. 10 summarizes the results of our key bandwidth optimization techniques for large local FFTs described in Section 5.2. The baseline implementation of Bailey's 6-step algorithm is denoted as `6-step-naïve`, which has 13 memory sweeps (Fig. 4(a)). An optimized implementation of the 6-step algorithm is denoted as `6-step-opt` (Fig. 4(b)). These two implementations already include vectorization and ILP optimizations presented in Section 5.2, and Fig. 10 highlights the impact of bandwidth-related optimizations. We apply two additional architecture-aware optimizations: memory latency hiding by prefetching and pipelining with SMT (`latency-hiding`), and fine-grain parallelization (`fine-grain`).

The performance of the final FFT implementation, 120 GFLOPS corresponds to ∼12% of compute efficiency. Assuming the number of memory sweeps to be 5, including the core-to-core communication (Section 5.2), the bops of large 16M FFT is $\frac{5\times16M\times16}{5\times16M\times\log_2 16M} = 0.67$, leading to $\approx$ 23% efficiency, similar to the one projected in Section 5.2. Our realized efficiency is ∼50% of this upper bound. Our implementation scales almost linearly with increase in cores; we obtain a speedup of ∼13 on 60 cores compared with 4 core executions, which leads to only 14% loss in compute efficiency.

We discover other two reasons that contribute to larger performance losses. First, computation is not entirely overlapped with memory transfers. We measure that steps that do not access main memory (e.g., step 2 in Fig. 4(b)) account for 36% of the total execution time. Second, bandwidth is not fully utilized due to long stride access. Steps 1, 4, and 6 access data in long strides that are comparable to the page size. This leads to TLB misses, which reduces the memory bandwidth efficiency of these steps as low as 50%.

6.3 Convolution-and-Oversampling

Fig. 11 demonstrates the impact of our key bandwidth optimization techniques for convolution described in Section 5.3. To highlight the impact of bandwidth-related optimizations, `baseline` is with vectorization, loop tiling, and unroll-and-jam optimizations applied. The loop interchange optimization is applied in `interchange` to reduce working set size, while using non-temporal stores. Circular buffers are used in `buffering` to reduce cache conflict misses by translating non-contiguous accesses to contiguous ones.

The loop interchange optimization not only helps the scalability but also the performance when only a small number of nodes are used. This is because applying thread-level parallelization to `loop_a` as shown in Fig. 7 reduces data sharing among cores, which is beneficial on Xeon Phi with private LLCs. In Xeon, we observe that the loop interchange optimization slightly degrades the performance of convolution in 4 nodes. Other than this, the impact of optimizations on Xeon is similar to that of Xeon Phi shown in Fig. 11 (recall that the same set of optimizations are applied to both Xeon and Xeon Phi). This optimization leads to a better Xeon performance than our previously reported one in [32], when measured in the same Endeavor cluster. Fig. 11 shows that `buffering` is also necessary to achieve close-to-ideal scalability. Without buffering, as we add nodes, the stride of accesses to the convolution input increases, causing more cache conflict misses.

7. COPROCESSOR USAGE MODES

This paper has only considered Xeon Phi's symmetric mode so far for brevity, but FFT can be called from applications written in either of symmetric and offload mode. This section briefly discusses implementation of multi-node 1D SOI FFT in offload mode. Local FFT and convolution implementations described in Sections 5.2 and 5.3 can be used in offload mode without modification. The difference is how data are communicated over PCIe between the host and Xeon Phi coprocessors.

Fig. 12 shows the timing diagrams of both modes. In

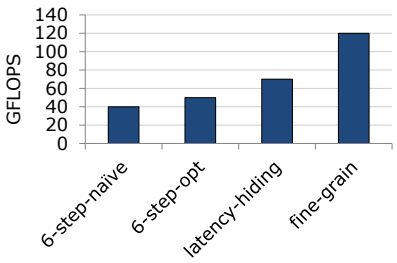


Figure 10: The impact of optimizations presented in Section 5.2 on 16M-point local FFT performance using a single Xeon Phi card

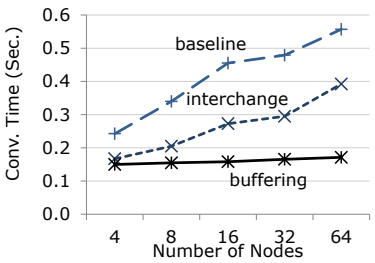


Figure 11: The impact of optimizations presented in Section 5.3 on convolution-and-oversampling execution time in Xeon Phi

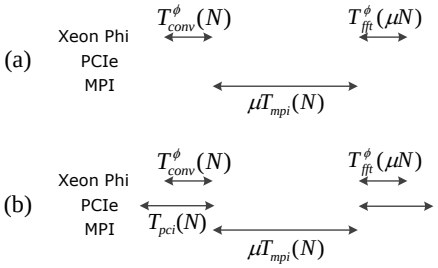


Figure 12: Timing diagrams of SOI FFT using Xeon Phi (a) in symmetric mode and (b) in offload mode.

both modes, PCI transfers for MPI communication can be completely hidden by overlapping it with InfiniBand transfers (Section 5.1). In offload mode, additional PCI transfers are required because inputs are not available in the memory of Xeon Phi and outputs need to be copied to the host memory. Due to the high compute capability of Xeon Phi, the local FFT and convolution are typically faster than each PCIe transfer, rendering PCIe transfers the bottleneck. Therefore, the execution time of Xeon Phi in the offload mode can be modeled as

$$T^{\phi_{\text{off}}}_{\text{soi}}(N) \sim 2T_{\text{pci}}(N) + \mu T_{\text{mpi}}(N),$$

where $T_{\text{pci}}(N)$ denotes the PCIe transfer time for N elements.

Assuming 6 GB/s PCIe bandwidth and the settings used in Section 4, Xeon Phis in offload mode are expected to be ∼25% slower than those in symmetric mode. Since we typically call FFT as a subroutine, the choice of coprocessor mode is often dictated by the application. However, when an application that will invoke large 1D FFTs frequently is being designed, our performance model can guide to select the right coprocessor usage mode.

The performance model presented in Section 4 can also be extended to a hybrid mode, where Xeon and Xeon Phi are used together. In the hybrid mode, Xeon Phi can be used in symmetric or offload mode. Even though Xeon processors provide additional compute capability in the hybrid mode, it is not evaluated in this paper. This is because only less than 10% speedups are expected from the additional compute due to the bandwidth-limited nature of 1D-FFT.

For brevity, it is assumed that MPI communication is not overlapped with computation in our performance model, while our actual implementation overlaps MPI communication time (T_{mpi}) with the local FFT computation time (T^{ϕ}_{fft}) as described in Section 6.1. However, it is straightforward to extend the model to consider the overlapping.

8. RELATED WORK

8.1 Low-Communication FFT Algorithms

Na'mneh et al. propose a no-communication parallel 1D FFT algorithm [22]. Their approach re-writes the Cooley-Tukey formulation using matrix vector products. Like our SOI algorithm, their main goal is to reduce (or eliminate) the communication steps. However, the asymptotic complexity of the algorithm is $O(N^{1.5})$, making it very compute intensive thus ineffective for large data size, while the SOI algorithm still remains as $O(N \log(N))$. The same authors also proposed a different variant two-step FFT algorithm in [21], where they could reduce the computational complexity at the cost of extra memory requirement. References to other low-communication FFT algorithms can be found in [32].

8.2 Large-Scale 1D FFT

A popular FFT algorithm for distributed memory machines is the 6-step algorithm [5] described in Section 5.2. Recently, Takahashi et al. have implemented this algorithm for K computer [31]. It uses the 6-step algorithm recursively to utilize the cache memories effectively, achieving a performance of over 18 TFLOPS on 8K nodes. The K computer was also awarded the first place in G-FFT of the 2012 HPCC rankings achieving 206 TFLOPS with 81K nodes [2]. However, the 6-step algorithm still requires 3 all-to-all communication. Even with the highly optimized Tofu interconnect [4], the all-to-all communication accounts for ∼50% of the total execution time. Doi et al. have implemented the 6-step algorithm on Blue Gene/P [10]. They pipeline all-to-all communication, overlapping computation with communication. They report 2.2 TFLOPS using 4K nodes of Blue Gene/P for 1D FFT of size 2^{24} complex double-precision numbers.

8.3 3D FFT on Clusters of Coprocessors

There are no known multi-node 1D FFT implementations on coprocessors. However, multi-GPU 3D FFT implementations have been presented. Chen et al. optimized a $4K^3$ 3D FFT on a 16-node GPU cluster, demonstrating how to utilize GPUs effectively for communication-intensive FFT [7]. McClanahan et al. implemented a 3D FFT on a 64-node GPU cluster and found that performance is more sensitive to intra-node communication governed by memory bandwidth rather than inter-node all-to-all communication [20]. On the contrary, Nukada et al. achieved a good strong scalability up to 4.8 TFLOPS on a 256-node GPU cluster emphasizing that efficient all-to-all communication between GPUs is the most important factor [23]. This paper tackles 1D FFTs that are more challenging than multi-dimensional FFTs that are less communication bound. We expect to see demonstrations of GPU clusters' performance on the more challenging 1D FFT via performance programming methodology similar to ours, for example, by applying our low-communication algorithm.

8.4 Node-Local 1D FFT on Coprocessors

There are several coprocessor implementations of large 1D FFT on a single node. Similar to our local 1D FFT implementation, they focus on reducing the number of memory sweeps in bandwidth-bound FFT computation, and are essentially variations of the 6-step algorithm. Chow et al. discuss an implementation of 2^{24}-point single precision FFT on Cell Broadband Engine [8]. They focus on reducing memory sweeps by performing the FFT in 3 mega-stages, where each mega-stage combines eight stages of radix-2 butterfly computation, thus requiring 6 memory sweeps overall. Naga et al. [14] present an implementation of large FFTs on NVIDIA's GTX280. Using an optimized 6-step algorithm with 4 memory sweeps, they show 90 GFLOPS on a card with peak performance of 936 GFLOPS (∼10% efficiency). Similar to ours, these local 1D FFT implementations for GPUs optimize bandwidth extensively, but we target Xeon Phi coprocessors with large on-chip caches, where different flavors of locality optimizations are desired.

9. CONCLUSION

We demonstrate the first tera-scale 1D FFT performance on Xeon Phi coprocessor. We achieve 6.7 TFLOPS on an 512-node Xeon Phi cluster, which is 5× better per-node performance than today's top HPC systems. It is enabled by our disciplined performance programming methodology that addresses the challenge of bandwidth requirements in 1D FFT. We systematically tackle the bandwidth issue at multiple levels of emerging HPC systems. For inter-node communication, we applied our low-communication algorithm. For PCIe transfers, we effectively overlap them with the inter-node communication. For memory accesses, we presented bandwidth optimization techniques, many of which are applicable to multiple compute kernels (local FFT and convolution-and-oversampling) and multiple architectures (Xeon Phi and Xeon).

Acknowledgements

The authors would like to thank Evarist M. Fomenko and Dmitry G. Baksheev for their help on Intel MKL performance evaluation, and Vladimir Petrov for his initial SOI FFT implementation that led to our earlier publication in last year's supercomputing conference. We also thank Stampede administrators in Texas Advanced Computing Center for advising tuning of MPI parameters to achieve the best network bandwidth. We also thank Intel Endeavor team for their tremendous efforts for resolving cluster instability in early installations of new hardware and software and for providing us exclusive access to a large set of nodes so that help our performance debugging.

References

[1] Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual.

[2] HPC Challenge Benchmark Results. `http://icl.cs.utk.edu/hpcc/hpcc_results.cgi`.

[3] RIKEN Next-Generation Supercomputer R&D Center. `http://www.nsc.riken.jp/index-eng.html`.

[4] Yuichiro Ajima, Yuzo Takagi, Tomohiro Inoue, Shinya Hiramoto, and Toshiyuki Shimizu. The Tofu Interconnect. In *Symposium on High Performance Interconnects (HOTI)*, 2011.

[5] David H. Bailey. FFTs in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23–35, 1990.

[6] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing Communication in Numerical Linear Algebra. *SIAM Journal of Matrix Analysis and Applications*, 32:866–901, 2012.

[7] Yifeng Chen, Xiang Cui, and Hong Mei. Large-Scale FFT on GPU clusters. In *International Conference on Supercomputing (ICS)*, 2010.

[8] Alex Chunghen Chow, Gordon C. Fossum, and Daniel A. Brokenshire. A Programming Example: Large FFT on the Cell Broadband Engine. In *Global Signal Processing Expo*, 2005.

[9] James W. Cooley and John W. Tukey. An Algorithm for the Machine Computation of Complex Fourier Series. *Mathematics of Computation*, 19(2):297–301, 1965.

[10] Jun Doi and Yasushi Negishi. Overlapping Methods of All-to-All Communication and FFT Algorithms for Torus-Connected Massively Parallel Supercomputers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[11] Franz Franchetti and Markus Püschel. *Encyclopedia of Parallel Computing*, chapter Fast Fourier Transform. Springer, 2011.

[12] Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. Discrete Fourier Transform on Multicore. *IEEE Signal Processing Magazine*, 26(6):90–102, 2009.

[13] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW. *Proceedings of the IEEE*, 93:216–231, 2005.

[14] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete fourier Transforms on Graphics Processors. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[15] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, and Pradeep Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel® Xeon Phi™ Coprocessor. In *IEEE International Parallel and Distributed Processing Systems (IPDPS)*, 2013.

[16] Balint Joo, Dhiraj D. Kalamkar, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Kiran Pamnany, Victor W Lee, Pradeep Dubey, and William Watson III. Lattice QCD on Intel R Xeon Phi coprocessors. In *International Supercomputing Conference (ISC), accepted for publication*, 2013.

[17] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. 2008. `www.cse.nd.edu/Reports/2008/TR-2008-13.pdf`.

[18] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transforms*. SIAM, 1992.

[19] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream`.

[20] Chris McClanahan, Kent Czechowski, Casey Battaglino, Kartik Iyer, P.-K. Yeung, and Richard Vuduc. Prospects for scalable 3D FFTs on heterogeneous exascale systems. 2011.

[21] R. Al Na'mneh and D. W. Pan. Two-step 1-D fast Fourier transform without inter-processor communications. In *Southeastern Symposium on System Theory*, 2006.

[22] R. Al Na'mneh, D. W. Pan, and R. Adhami. Parallel implementation of 1-D Fast Fourier Transform without interprocessor communication. In *Southeastern Symposium on System Theory*, 2005.

[23] Akira Nukada, Kento Sato, and Satoshi Matsuoka. Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[24] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Küger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[25] Jongsoo Park, Ping Tak Peter Tang, Mikhail Smelyanskiy, Daehyun Kim, and Thomas Benson. Efficient Backprojection-based Synthetic Aperture Radar Computation with Many-core Processors. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[26] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *International Conference on Management of Data (SIGMOD)*, 2010.

[27] Fengguang Song, Hatem Ltaief, Bilel Hadri, and Jack Dongarra. Scalable Tile Communication-avoiding QR Factorization on Multicore Cluster Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[28] Daisuke Takahashi. A Parallel 3-D FFT Algorithm on Clusters of Vector SMPs, 2000.

[29] Daisuke Takahashi. A parallel 1-D FFT algorithm for the Hitachi SR8000. *Parallel Computing*, 29:679–690, 2003.

[30] Daisuke Takahashi, Taisuke Boku, and Mitsuhisa Sato. A Blocking Algorithm for Parallel 1-D FFT on Clusters of PCs. In *International Euro-Par Conference*, number 2400, 2002.

[31] Daisuke Takahashi, Atsuya Uno, and Mitsuo Yokokawa. An Implementation of Parallel 1-D FFT on the K Computer. In *International Conference on High Performance Computing and Communication*, 2012.

[32] Ping Tak Peter Tang, Jongsoo Park, Daehyun Kim, and Vladimir Petrov. A Framework for Low-Communication 1-D FFT. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[33] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.